Graph Construction
In the Processing Graph Method Tool

by
Dick Stevens
September 9, 2002

1 Introduction

At NRL (Naval Research Laboratory) we wrote the PGM (Processing Graph Method) Specification [1] to describe the basic features of a support system for developing applications for a distributed network of processors. In the PGMT (PGM Tool) project we developed an implementation of PGM. This project was conducted with the support of the Advanced Systems Technology Office for Undersea Warfare in the Office of Naval Research.

We assume the reader is familiar with the PGM Spec [1], the GUI user manual [2], and Implementation of Families [3].

In this paper we describe the various structures and procedures used in PGMT for constructing a processing graph. Our approach is to construct and maintain two levels of the processing graph.

The first level, the *hierarchical graph* comprises the elements that the signal processing engineer identified while using the GUI to define the processing graph. These elements are the families of graph ports as well as nodes and included graphs, together with their respective families of node ports and included graph ports. Each family has a name, and each family member has the associated family indexing. In this higher level of the graph structure, we use what we call a *handle* to represent each family, be it a family of graph ports, a family of included graphs, or a family of nodes. Each graph and each included graph has a list of the handles of its included graph ports, as well as its nodes and included graphs. The handle for a family of included graphs has a family descriptor and an array of identifiers for the respective included graphs, thus providing a hierarchical structure for the entire processing graph. Connections between ports are represented in terms of the family names and family indexing.

The second level, the *flat graph*, comprises the actual processing elements of the processing graph. These elements are the nodes of the graph; i.e., the transitions and places that do the processing. The included graphs have been expanded to their component nodes. Likewise, the families have been expanded to their components. Families of nodes, families of graph ports, families of included graphs, and included graphs, themselves, do not exist in the flat graph. Only the graph ports and the nodes exist. Connections between node ports are direct, so that each pair of connected node ports bypasses any intervening included graph ports.

While the flat graph supports efficient graph execution, we use the hierarchical graph as an interface for the user. Each node in the flat graph has the identifier for its family handle in the hierarchical graph as well as its index in the leaf array. These are used to construct a string giving the node's family name and

family indices. If the handle lies in an included graph, then the full name of the node is obtained by appending the node's family name and family indices to the full name of the included graph. The full name of the included graph is obtained recursively in the same way, by appending the full name of its parent graph to its family name and family indices. This provides a means by which the node can be identified for the user.

Another advantage to this approach is that the hierarchical graph provides a representation of the graph that can be used to make a copy of the graph. In PGMT we did not implement this feature; however the ability to copy a graph was a design consideration.

In the next section we give an overview of PGM, including a description of command programs, graphs, and included graphs, emphasizing the relationship between node ports and graph ports. We then describe the various data structures that are used during graph construction for the flat graph as well as for the hierarchical graph.

2 Processing Graph Method (PGM)

In this section we give a more detailed description of PGM graphs. We begin by discussing the notion of a *family*, which occurs throughout PGM.

2.1 Families

A *family* in PGM extends the notion of a finite multi-dimensional array. Families are used extensively in PGM. For a full discussion of families, see the PGM Spec [1] and the Implementation of Families [3].

2.2 Tokens

The data elements that pass from one node to another via the directed arcs are called *tokens*. Each token is a family. The type of a token is its height together with its base type. Its family tree and the values of its respective leaves determine the value of a token. The base type of a token may be a standard variable type in the high-order language, like integer or float, or it may be a user-defined type.

2.3 Node Ports and Graph Ports

A PGM graph comprises a set of nodes and directed arcs. In each node, the point of connection with a directed arc is called a *node port*. Each port is either an *input port* or an *output port*. A node input port is connected to the *head* of a directed arc, and a node output port is connected to the *tail* of a directed arc. In this way, a directed arc points in the direction of the flow of data from one node to another.

All tokens passing from one node to another via a directed arc must have the same type. Thus we define the *mode* of a node port to be the common type of the tokens.

For a directed arc to connect two node ports, PGM requires the following:

- One of the ports must be an input port and the other an output port.
- One of the ports must be a transition port and the other a place port.
- Both ports must have the same mode.

Tokens enter a graph via one or more input ports of nodes in the graph. Similarly, tokens leave a graph via output ports of nodes in the graph. Thus we define the notions of *graph input ports* and *graph output ports*. A graph input (output) port is associated with an input (output) port of either a node in the graph or an included graph in the graph. This being a recursive definition, we see that a graph input (output) port is associated ultimately with a node input (output) port at some level of nested included graph. Each graph port has the same attributes as its associated node port:

- Direction (input or output),
- Category (transition or place), and
- Mode.

We support the construction of a family of node ports. PGM stipulates that all ports in the family have the same mode. For example, it is possible that a given node may have a family of node ports. Another possibility occurs when there is a family of nodes, each having a single node port. The aggregation of the ports across the family of nodes comprises a family of ports. It is also possible that each node in a family of nodes may have a family of ports. The family tree of the resulting family of node ports is assembled as described in the above section on families.

In any of the above examples, the family of node ports may be associated with a family of graph ports. The family tree of the graph port family is the same as the assembled family of associated node ports.

In a similar fashion, a family of graph ports may be associated with a family of included graph ports in which there is a family of included graphs, each having a family of graph ports. As above, the family tree of the graph ports is the same as the assembled family of associated included graph ports.

2.4 Command Program, Main Graph, and Included Graphs

A user application has two major components. The first component is the *command program*. The user writes the command program in a standard high-order language that is specific to the PGM implementation. In PGMT, the command program language is C++. The PGM implementation provides a set of functions that the command program may call to for various services to construct, interact with, and destroy the graph.

The second major component is the PGM graph, which we refer to as the *main graph*, or more simply, *the graph*. To facilitate the specification of the graph, PGMT provides a GUI (Graph user interface), which allows a human user to construct a graphic image of the graph using icons to represent the nodes and arrows to represent the directed arcs. The GUI also provides tables associated with each icon for the specification of text information to specify various parameters. When the GUI specification is complete, the user may request that code be automatically generated for a *graph construction procedure*. When the command program calls this procedure, it creates the PGM graph in the application architecture and prepares the graph for execution. For a full description of the GUI, see the GUI User Manual [2].

Beyond representing a node, a given icon on the GUI screen may also represent an *included graph*, which is another graph that may itself be specified using the GUI. The graph construction procedure for the main graph will call the graph construction procedure for the included graph. Each call of the included graph construction procedure creates a new instance of the included graph. The construction procedure for an included graph may, in turn, call another included graph construction procedure. However, no graph construction procedure may call itself either directly or indirectly. If the icon represents a family of included graphs, the construction procedure for that included graph is called repeatedly – once for each leaf in the family.

A given icon may also represent a family of nodes or a family of included graphs. By convention, in PGMT, every icon represents a family of nodes or a family of included graphs. The default family height is 0, in which case the icon represents a single node or included graph. For each icon, the user has the option of specifying any family tree of any height. Each leaf in the family is a separate instance of the node or included graph.

When the main graph is fully constructed, each node has a unique identification that is based on names and family indexing that the user specified. Each icon has a name that is unique within its graph. We call this name the *family name*. A specific node in a main graph or included graph is identified by its family name and family indices. If a node is specified in an included graph (which may be one of a family of included graphs), the node is identified by the name and family indices of the included graph, followed by the name and family indices of the node in its included graph. We use this technique recursively to provide an identification scheme for every node in the graph.

Because a given icon may represent multiple nodes or included graphs, a given directed arc on the GUI screen may also represent multiple connections between node ports and/or included graph ports. In a similar manner, a single node or a single included graph may have a family of ports. In this case as well, a directed arc on the GUI screen may represent multiple connections between ports.

An included graph port may be associated with either a place port or a transition port. However a main graph port must be associated with a place port. The command program has access only to the graph ports of the main graph.

The command program produces input tokens to the graph via its graph input ports by calling standard PGM procedures. This results in the storing of the input tokens in respective queues in the graph. Similarly the command program reads output tokens from the graph via its graph output ports by calling standard PGM procedures. This results in the reading and consumption of output tokens from respective queues in the graph.

2.5 Nodes

As stated above, each node is either a transition or a place. Places store tokens, and transitions perform processing. A transition reads tokens via its input ports from places that are connected by directed arcs. After performing calculations, a transition produces new tokens via its output ports to be stored in places that are connected by directed arcs.

During graph construction, each node is constructed by calling its respective construction procedure.

For a more complete discussion of transitions and places, see the PGM Spec [1].

In the PGMT, the language is C++. The user-defined main graph and all included graphs, as well as all nodes, are defined by classes. The construction procedure for each of these is a constructor of the respective class. In a family of nodes or of included graphs, every family member is a member of the same class and thus is constructed from the constructor of that class.

3 Graph Construction

After the PGM graph is constructed, it is the nodes and the connections between their ports that determine the processing. One major goal during graph construction is to expand all the families of nodes and families of included graphs and to make all the connections directly between node ports, leading to a graph structure that contains only individual nodes with connections between ports. This will allow transitions and places to have direct access to each other rather than having to resolve node family identities and association of included graph ports during every execution of a transition. The PGM graph so constructed is called the *flat graph*, and the process of constructing the flat graph is called *graph flattening*.

While constructing the flat graph, we wish to retain all of the included graph and family structure with which the graph writer is familiar, so that a particular node can be identified in terms that the user will

understand. This desire led to the *hierarchical graph*, which is an additional set of structures that support the tracking of the family and included graph structure defined by the graph writer.

Graph construction proceeds recursively. If the graph has no included graphs, the graph construction procedure constructs all of the nodes, using loops to expand families of nodes. The graph construction procedure then connects the ports of nodes within the graph. Finally, all graph port families are associated with their respective internal node port families. Both the hierarchical graph and flat graph are constructed in this procedure.

If the graph has an included graph, then the graph's construction procedure constructs each included graph by calling the respective graph construction procedure. As with a family of nodes, a loop is used to make repeated calls for a family of included graphs. When the included graph construction is complete, all of its internal structures for both the hierarchical and flat graphs are complete, including the connections of all node ports within the included graph. The graph port families of the included graph are associated with the respective internal node port families or internal included graph port families.

To facilitate the introduction of tokens into the graph via its graph input ports and the retrieval of tokens from a graph via its graph output ports, we augment the graph by constructing a *pseudo-transition* for each graph port. For each family of graph ports, we construct a family of pseudo-transitions.

To pass an input token into the graph via a graph input port, the command program calls an appropriate procedure. Likewise, to retrieve an output token via a graph output port, the command program calls another procedure. Whereas the PEP controls the execution of a transition in the PGM graph, the execution of a pseudo-transition is triggered by a call from the command program. Even though these pseudo-transitions are not part of the user's graph, we consider them to be nodes in the augmented graph and thus include them in the graph construction process.

It is important to note that the pseudo-transitions are constructed only for main graph ports and not for included graph ports.

We now discuss the data structures used during graph construction. We discuss construction of both the hierarchal graph and the flat graph. Following that we will discuss graph execution.

3.1 Family and Included Graph Hierarchy

In what follows, many of the data structures contain machine addresses of other data structures. We will speak of a *pointer* to a data structure to mean the machine address of that data structure.

To keep track of the hierarchal graph structures, each graph (main graph and included graph) maintains its own list of families of nodes, families of included graphs, and families of graph ports. The data structure for each family is called a *handle*, and the list of handles is called the *graph handle table*. The graph handle table contains a pointer to each handle, and we will use the index in the graph handle table to identify the handle.

Each handle contains the user-specified family name, a descriptor for the family, and a data array of pointers to the individual leaves in the family, be they nodes, included graphs, or graph ports.

The data structure for each leaf contains a list of the input port handles and a list of output port handles, there being a port handle for each port family. Each port handle contains the user-specified port family name, the category of the port (i.e., either *transition* or *place*, to support the checking of correctness that a place port can only be connected to a transition port and vice versa), a descriptor for the port family, and a data array of *port finders*. Each port finder corresponds to a single port P, and the port finder is an array of four integers to identify the port to which P is connected.

The first integer in the port finder is the index in the graph handle table to identify the handle of the family (node, included graph, or graph port). The second integer is the index in the data array to identify the family member. The third integer is the index into the respective input or output port handle table of the individual node, included graph, or graph port. Whether it is an input port or output port is resolved by the requirement that an input port can be connected only to output port and vice versa. Finally, the fourth integer identifies the individual port in the port family.

Figure 1 shows a schematic view of the data structures in the hierarchical graph. The graph handle table at the upper left is a table containing pointers to the handles. Each handle contains the following information:

- The object type (transition, place, included graph, graph port).
- The string IDs (a string identifying the object) This is primarily for use by the PEP.
- The index (i.e., the handle's index in the graph handle table).
- The descriptor for the family in the handle.
- The object information table (pointer to a table of the leaves in the family).
- The input port information table (pointer to a table of information about the input ports).
- The output port information table.

Each entry in the object information table is a pointer to the information block about each leaf in the family. This information block contains three pointers:

- A pointer to the table of input port handles (each handle representing an input port family).
- A pointer to the table of output port handles.

- A pointer to the leaf (the leaf being of the type identified above in the object type entry in the graph object handle – i.e., transition, place, included graph, or graph port).

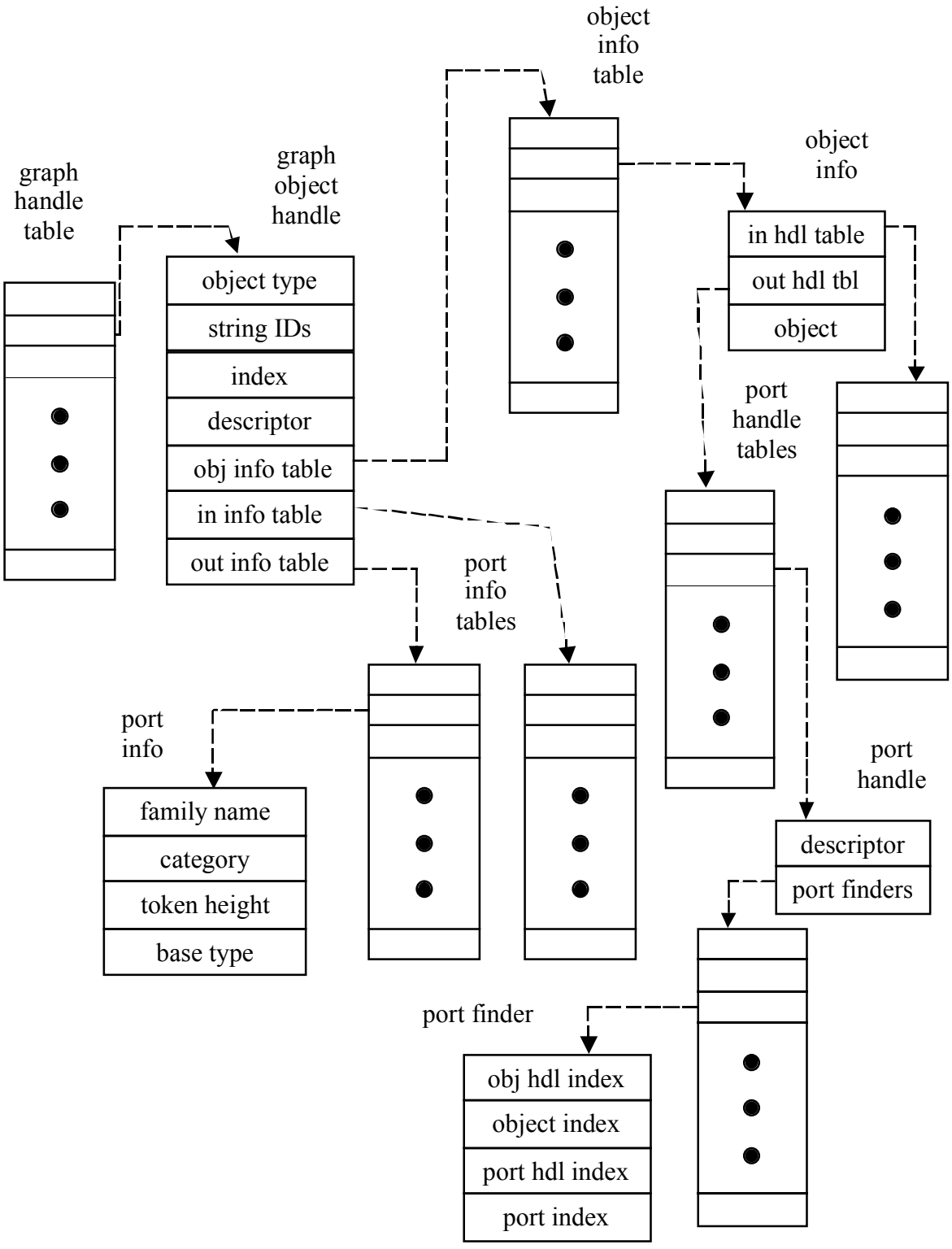

Figure 1

If the leaf is a node (i.e., transition, or place), then the pointer is the address of the respective node's data structure. If the leaf is an included graph, then the pointer is the address of the respective included graph object, which contains its own graph handle table. If the leaf is a graph port, and the graph is the main graph, then the leaf is the pseudo-transition.

In the port handle table, each entry is a pointer to the port handle. The port handle, which represents a port family, contains a descriptor and a table of port finders – one port finder for each port in the port family. As explained above, the port finder for a given port P, is a 4-tuple of indices to identify the port to which P is connected.

In the graph object handle, the last two entries are pointers to the input port and output port information tables. Each entry in one of these tables identifies the following attributes about the respective port family. Note that these attributes hold for every port in the family:

- The port family name.
- The category of the port (transition or place).
- The token height (the token height and base type represent the mode of the port).
- The token base type (the base type here is the MPI type, as determined by MTOOL).

Immediately following the construction of each handle, the various tables described above are constructed and filled out with the appropriate information. In particular, each leaf is constructed and its address inserted into the object information block. If the leaf is an included graph, this means that the included graph construction procedure is called. In a recursive fashion, the entire included graph is constructed before we proceed with the next handle.

3.2 Flat Graph Construction

During the process of constructing handles in the hierarchical graph as described above, every node in the flat graph is constructed, complete with all of the data needed for processing during graph execution. For each handle, all of the members of its family are constructed during this time. In the main graph there is a *node table*. As each node is constructed in the main graph or in any included graph, an entry is made in the node table. Thus, during graph execution, the node table lists all the nodes in the flat graph. If a function must be applied to all nodes in the flat graph, the node table can be used to access all the nodes. Note that this node table includes all nodes that were constructed in included graphs at all levels.

After the handle table is complete, connections are made between node ports in the flat graph. To do this, the port finders are constructed. The node or included graph family name identified by the user is converted to an index in the graph handle table. The family indices are converted to a data array index to identify the included graph or node in the family. Then in a similar fashion, the port family name is converted to an index in the node's respective input or output port handle table. Finally, the port family indices are converted to a data array index to identify the port. These four indices are placed in the port finder, which enables compact storage as described above.

To enable quick searches for names to identify handles in the graph handle table, the graph handle table is alphabetized after all handles have been constructed and before any port connections are made. After the graph handle table is alphabetized, each handle may be identified by its index in the graph handle table, and each handle stores its own index in the graph handle table.

If the port finder identifies a port of an included graph (i.e., if the first integer in the port finder is the index of a handle for a family of included graphs), then the associated node port must be identified. The function that does this is recursively defined, because the included graph port may itself be associated with a graph port of another included graph nested within. There may be several levels of included graphs before the node port is identified.

The above process resolves all of the included graph structure, so that intermediate ports of included graphs are bypassed. Thus direct connections between respective node ports in the flat graph are made once and for all during graph construction, reducing run-time overhead and enhancing graph execution performance.

As mentioned above, the node table contains a list of every individual node in the entire graph. Each node contains a list of its input port families and a list of its output port families. Each port family in the node has a descriptor for its family and an array of addresses of, or *pointers to*, the individual ports. Each port has a pointer to the node port to which it is connected.

As each connection between node ports is made, the respective pair of nodes is added to the node pair list. This list will be passed to the PEP. The PEP, in turn, will use this list to determine the topology of the PGM graph when determining the initial assignment and subsequent reassignments.

As stated above, the flat graph comprises just the nodes and connections between their ports. With very few exceptions, no reference is needed to any of the handles during normal graph execution. To determine assignment of transitions to processes and to schedule transitions for execution, the PEP refers directly to the flat graph and to the node pair list.

As a final note, every node in the flat graph maintains a pointer to its handle as well as its data array index in the family that its handle represents. In this way, if a problem occurs, the node name and family indices (if any) can be obtained for the user. Indeed, the full name (including the included family hierarchy) can be obtained as well.

4 Conclusion

In this paper we have given the high-level design for graph construction in PGMT, comprising the hierarchical graph and the flat graph.

The hierarchical graph comprises the data structures that represent the graph structure as defined by the GUI user. To manage these data structures, the main graph and each included graph maintains a list of handles that represent the families of nodes and included graphs. Ancillary data structures are provided to support validation of port connections and navigation within the hierarchical graph.

The flat graph comprises data structures, or class objects, for the transitions and places in the graph that are used during graph execution.

Graph execution occurs in the flat graph. If a problem occurs, each node can identify itself to the user by using its access to its handle and its data array index in the family. Each handle, in turn, has access to its immediate parent graph (main graph or included graph) and its own index in the handle table. Each included graph has access to its immediate parent graph as well as its handle and data array index in the handle table of that parent graph.

5 References

1. *Processing Graph Method 2.1 Semantics*, by David J. Kaplan and Richard S. Stevens, Naval Research Laboratory, Washington, DC, April 1, 2001.
2. *GUI User Manual* (Exact Title TBS), by Michal Iglewski, Naval Research Laboratory, Washington, DC, (Date TBS).
3. *Implementation of Families*, by Dick Stevens, Naval Research Laboratory, Washington, DC, August 29, 2002.